

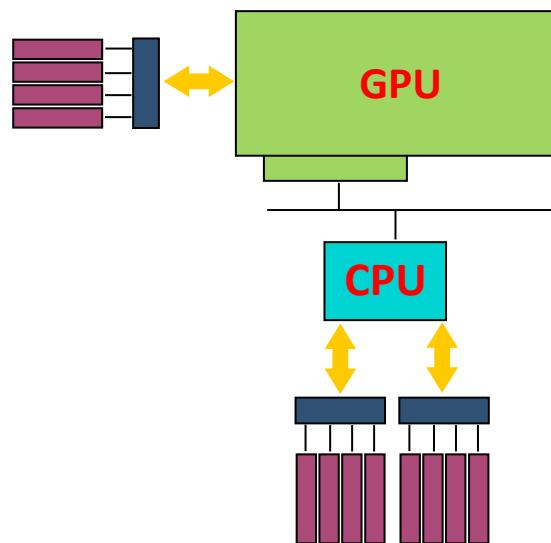
Towards Performance Portable GPU Programming with RAJA

Arpit Jacob, Samuel Antao, Hyojin Sung, Alexandre Eichenberger,
Carlo Bertolli, Gheorghe-Teodor Bercea, Tong Chen, Zehra Sura,
Georgios Rokos, Kevin O'Brien

IBM T. J. Watson Research Center

Performance Portability Problem

- Applications will have to exploit heterogeneous resources in a performance portable manner



- Using vendor specific languages and directives
- Compiler specific pragmas
- Mix of programming models: OpenMP, OpenACC, CUDA

The RAJA Approach

- RAJA* is a C++ based programming approach that
 - Hides computing paradigm and parallel programming models from user
 - With minimum code disruption
- Expresses parallelism in inner loops
- Uses C++11 Lambdas and templates

* R. D. Hornung, and J. A. Keasler. 2014. The RAJA Portability Layer: Overview and Status. LLNL-TR-661403. Lawrence Livermore National Laboratory

The RAJA Approach

- **Fundamental concepts:** traversal template, execution policy, loop body

```
for (int i=0; i<len; i++) {  
    bvc[i] = cls * (compression[i] + 1.0);  
}
```

C/C++ style loop

Traversal Template Policy

```
forall<simd_exec>(0, len,  
    [&] (int i) {  
        bvc[i] = cls * (compression[i] + 1.0);  
    }  
);
```

Body

RAJA loop

RAJA: SIMD Execution

- Exploit SIMD on CPU with the Clang compiler

```
template <typename LOOP_BODY>
inline
void forall ( simd_exec, int begin, int end,
              LOOP_BODY loop_body) {
    #pragma clang loop vectorize(enable)
    for (int i = begin; i < end; i++) {
        loop_body(i);
    }
}
```

RAJA: Hardware Threads with OpenMP

```
forall<cpu_parallel_exec>(0, len,  
    [&] (int i) {  
        bvc[i] = cls * (compression[i] + 1.0);  
    }  
);
```

```
template <typename LOOP_BODY>  
inline  
void forall ( cpu_parallel_exec, int begin, int end,  
             LOOP_BODY loop_body) {  
    #pragma omp parallel for  
    for (int i = begin; i < end; i++) {  
        loop_body(i);  
    }  
}
```

RAJA: Streaming Multiprocessors on a GPU

```
forall<gpu_parallel_exec>(0, len,  
    [&] (int i) {  
        bvc[i] = cls * (compression[i] + 1.0);  
    }  
);
```

<gpu_parallel_exec>

?

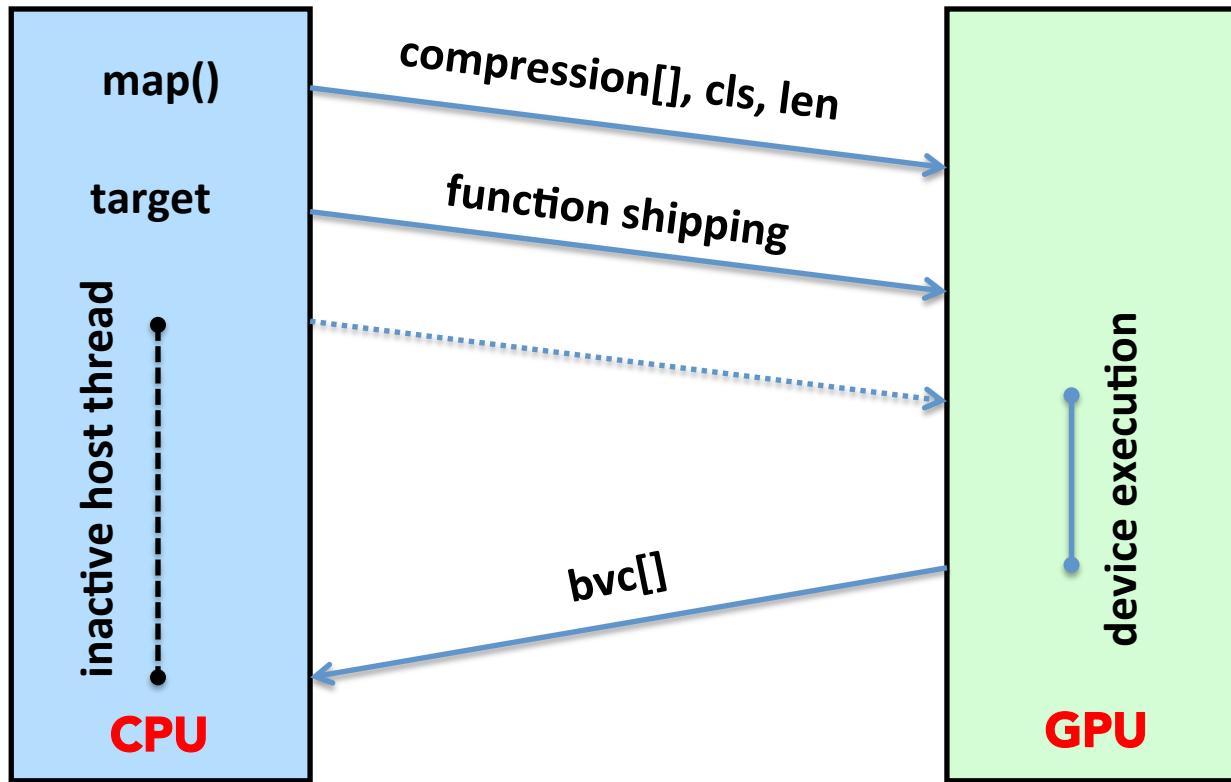
GPU Offloading with OpenMP 4.5

- Latest OpenMP specs support offloading to accelerators
- **How do we use OpenMP offload with RAJA?**

```
#pragma omp target map(to: cls, len, compression[0:len]) \
    map(from: bvc[0:len])
```

```
#pragma omp parallel for
for (int i=0; i<len; i++) {
    bvc[i] = cls * (compression[i] + 1.0);
}
```

GPU Offloading with OpenMP 4.5



```
#pragma omp target map(to: cls, len, compression[0:len]) \
map(from: bvc[0:len])
```

```
#pragma omp parallel for
for (int i=0; i<len; i++) {
    bvc[i] = cls * (compression[i] + 1.0);
}
```

First Try: RAJA with OpenMP 4.5

```
forall<gpu_parallel_exec>(0, len,
    [&] (int i) {
        bvc[i] = cls * (compression[i] + 1.0);
    }
);
```

```
template <typename LOOP_BODY>
inline
void forall ( gpu_parallel_exec, int begin, int end,
    LOOP_BODY loop_body) {
```

```
#pragma omp target
#pragma omp parallel for
for (int i = begin; i < end; i++) {
    loop_body(i);
}
```

Data variables of loop body
are not visible in template
scope

Lambda function is not GPU code

Mapping LAMBDAs in the Compiler

```
forall<gpu_parallel_exec>(0, len,
```

```
[&] (int i) {  
    bvc[i] = cls * (compression[i] + 1.0);  
}  
};
```

```
#pragma omp target  
for (int i = begin; i < end; i++)  
    loop_body(i);
```

map variables
captured
by lambda

```
struct <LAMBDA> {  
    double* &bvc;  
    double &cls;  
    double* &compression;
```

create GPU
function

```
void operator()(int i) {  
    bvc[i] = cls * (compression[i] + 1.0);  
}
```

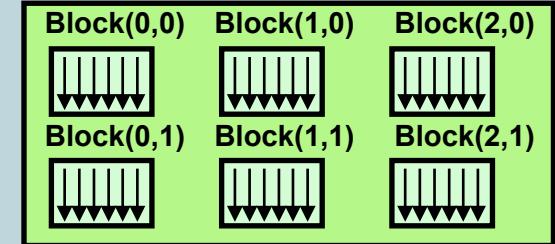
Establishing a GPU Data Environment

```
struct ADomain {
    ADomain(int ilen, int ndims ) {
        // Initialization code
        // Copy data onto the GPU
        #pragma omp target enter data \
            map(to: zones[0:n_zones])
    }
    UpdateHost() {
        // Retrieve from the GPU
        #pragma omp target exit data \
            map(from: zones[0:n_zones])
    }
    ~ADomain() {
        // Delete data from the device
        #pragma omp target exit data \
            map(delete: zones[0:n_zones])
        delete []zones;
    }
    ...
};

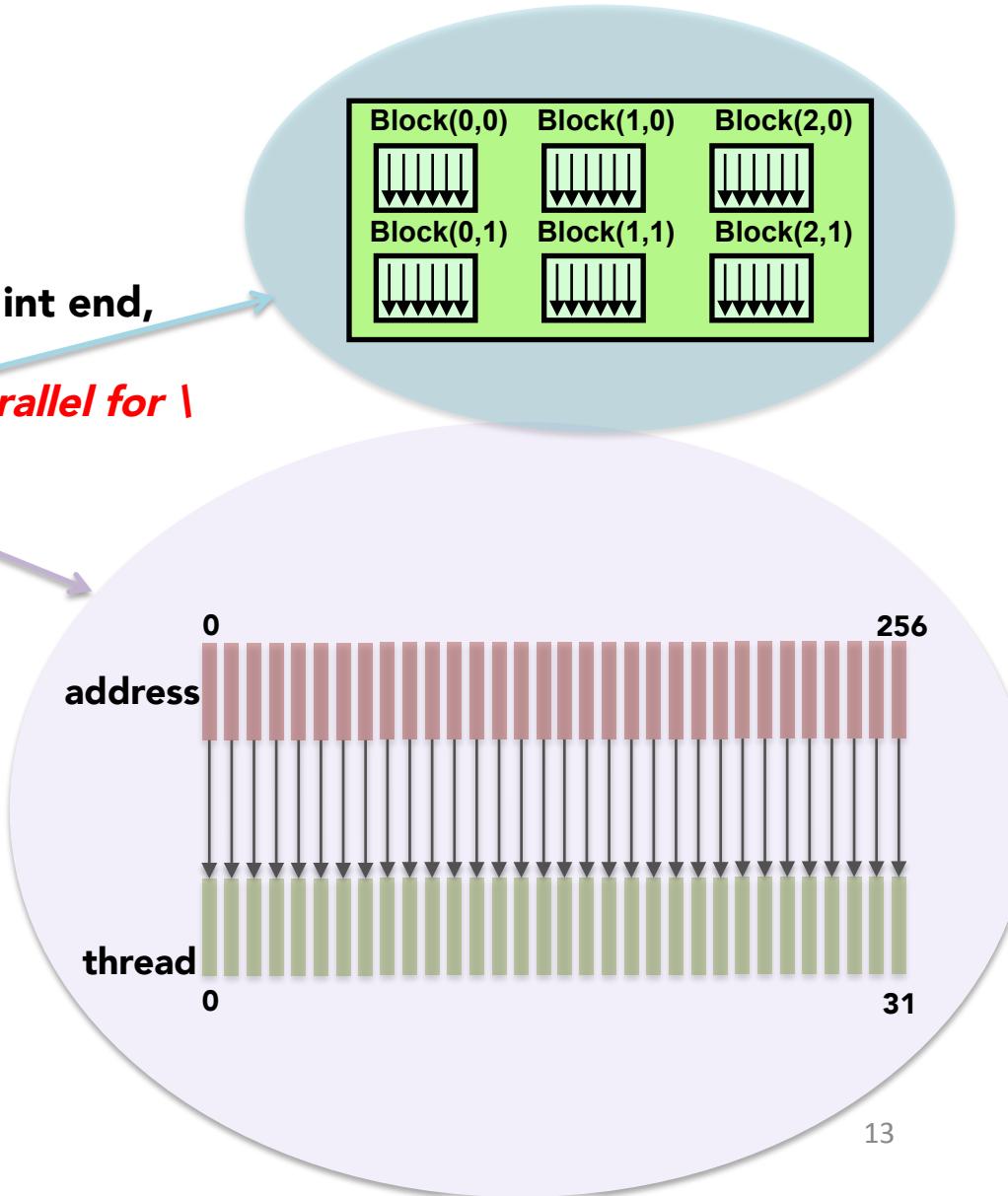
{ ADomain domain(LENGTH, 2);
...
forall<gpu_parallel_exec>()
forall<gpu_parallel_exec>()
domain.UpdateHost();
...
}
```

Optimizing RAJA OpenMP 4.5 Programs

```
template <typename LOOP_BODY>
inline
void forall ( gpu_parallel_exec, int begin, int end,
              LOOP_BODY loop_body) {
#pragma omp target teams distribute parallel for \
schedule(static, 1)
    for (int i = begin; i < end; i++) {
        loop_body(i);
    }
}
```



specialize parallel loop kernel call
to remove OMP overheads

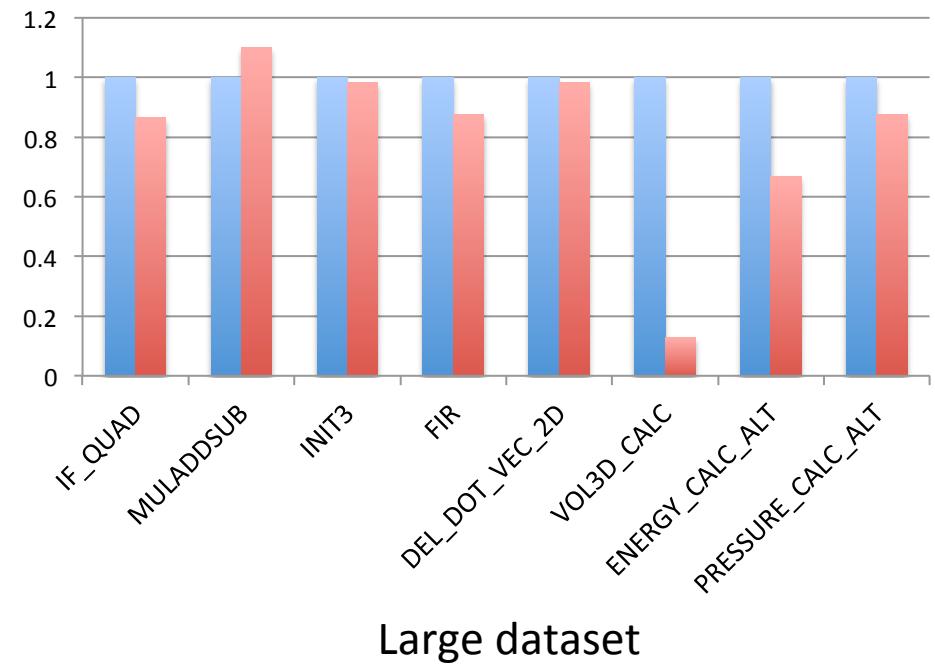
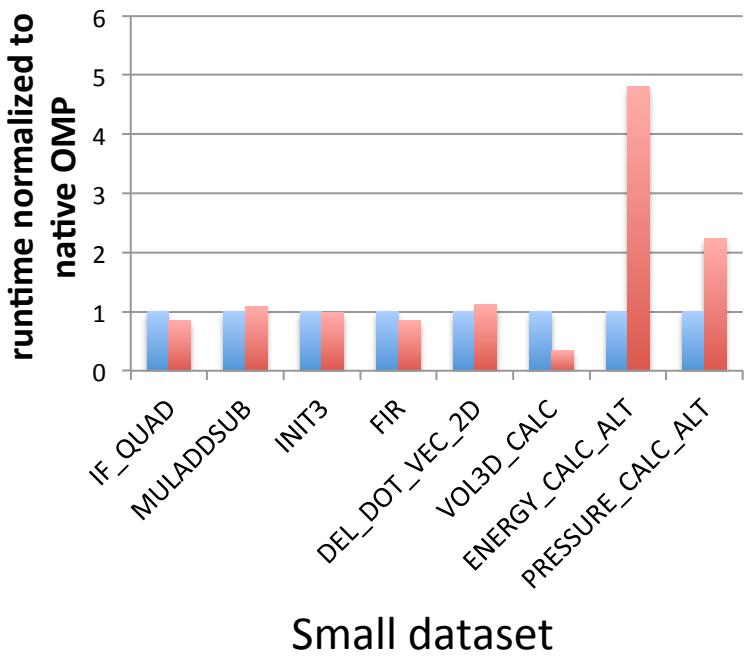


RAJA with GPUs

```
forall<gpu_parallel_exec>(0, len,  
    [&] (int i) {  
        ...  
    }  
);
```

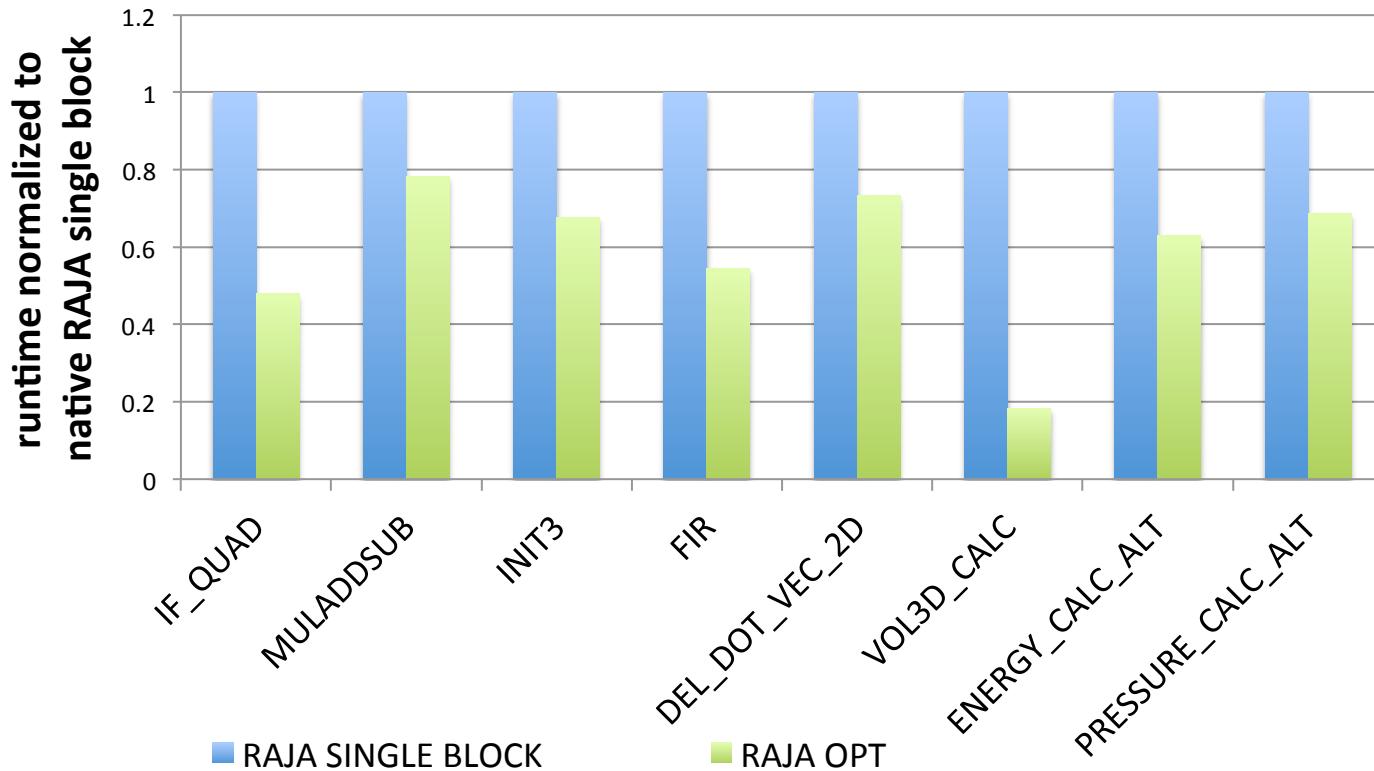
Results: RAJA vs. OMP

- Studied the LCALS benchmark, a collection of floating-point scientific kernels released by LLNL
- Ported to OMP4.5 and RAJA+OMP4.5
- Machine: NVIDIA Kepler K40m GPU and IBM Power8 CPU



Results: Impact of GPU-specific directives

- Baseline: Single thread block
- Optimized: Multiple thread blocks, minimized overhead



Conclusions & Future Work

- Extended RAJA for GPU execution with OMP4.5
- Required additional support from the compiler
- High performance requires GPU-specific OMP directives and clauses, but they can be hidden in RAJA
- Focus on inner loop has performance advantages
- Explore asynchronous offloading to reduce GPU invocation overhead
- Download and install LLVM/OpenMP 4.x compiler and runtime for NVIDIA GPUs from:

<https://www.ibm.com/developerworks/community/groups/community/openmp>

This work is partially supported by the CORAL project LLNS Subcontract No. B604142.